Modern C++ for Computer Vision and Image Processing

Igor Bogoslavskyi
Outline

**Smart pointers**
- Unique pointer
- Shared pointer

**Associative containers**

**Type casting**
- static_cast
- reinterpret_cast
- dynamic_cast

**Enumeration classes**

**Read/write binary files**
Smart pointers
Smart pointers

- Smart pointers wrap a raw pointer into a class and manage its lifetime (RAII)
- Smart pointers are all about ownership
- Always use smart pointers when the pointer should own heap memory
- **Only use them with heap memory!**
- Still use raw pointers for non-owning pointers and simple address storing
- `#include <memory>` to use smart pointers
- We will focus on 2 types of smart pointers:
  - `std::unique_ptr`
  - `std::shared_ptr`
Smart pointers manage memory!

Smart pointers apart from memory allocation behave exactly as raw pointers:

- Can be set to `nullptr`
- Use `*ptr` to dereference `ptr`
- Use `ptr->` to access methods
- Smart pointers are polymorphic

**Additional functions of smart pointers:**

- `ptr.get()` returns a raw pointer that the smart pointer manages
- `ptr.reset(raw_ptr)` stops using currently managed pointer, freeing its memory if needed, sets `ptr` to `raw_ptr`
Unique pointer (**std::unique_ptr**) 

- Constructor of a unique pointer takes ownership of a provided raw pointer
- **No runtime overhead** over a raw pointer
- Syntax for a unique pointer to type **Type**:

```
#include <memory>

// Using default constructor Type();
auto p = std::unique_ptr<Type>(new Type);

// Using constructor Type(<params>);
auto p = std::unique_ptr<Type>(new Type(<params>));
```

- **From C++14 on:**

```
// Forwards <params> to constructor of unique_ptr
auto p = std::make_unique<Type>(<params>);
```

http://en.cppreference.com/w/cpp/memory/unique_ptr
What makes it “unique”

- Unique pointer has no copy constructor
- Cannot be copied, can be moved
- Guarantees that memory is always owned by a single unique pointer

```cpp
#include <iostream>
#include <memory>
struct A {
  int a = 10;
};
int main() {
  auto a_ptr = std::unique_ptr<A>(new A);
  std::cout << a_ptr->a << std::endl;
  auto b_ptr = std::move(a_ptr);
  std::cout << b_ptr->a << std::endl;
  return 0;
}
```
Shared pointer (**std::shared_ptr**)

- Constructed just like a **unique_ptr**
- Can be copied
- Stores a usage counter and a raw pointer
  - Increases usage counter when copied
  - Decreases usage counter when destructed
- Frees memory when counter reaches 0
- Can be initialized from a **unique_ptr**

```cpp
#include <memory>

// Using default constructor Type();
auto p = std::shared_ptr<Type>(new Type);

auto p = std::make_shared<Type>(); // Preferred

// Using constructor Type(<params>);
auto p = std::shared_ptr<Type>(new Type(<params>));
auto p = std::make_shared<Type>(<params>); // Preferred
```

#include <iostream>
#include <memory>

struct A {
    A(int a) { std::cout << "I'm alive!\n"; }
    ~A() { std::cout << "I'm dead... :(!\n"; }
};

int main() {
    // Equivalent to: std::shared_ptr<A>(new A(10));
    auto a_ptr = std::make_shared<A>(10);
    std::cout << a_ptr.use_count() << std::endl;
    {
        auto b_ptr = a_ptr;
        std::cout << a_ptr.use_count() << std::endl;
    }
    std::cout << "Back to main scope\n";
    std::cout << a_ptr.use_count() << std::endl;
    return 0;
}
When to use what?

- Use smart pointers when the pointer **must manage memory**
- By default use `unique_ptr`
- If multiple objects must **share** ownership over something, use a `shared_ptr` to it
- Using smart pointers allows to avoid having destructors in your own classes
- Think of any free standing `new` or `delete` as of a memory leak or a dangling pointer:
  - Don’t use `delete`
  - Allocate memory with `make_unique`, `make_shared`
  - Only use `new` in smart pointer constructor if cannot use the functions above
Typical beginner error

```cpp
#include <iostream>
#include <memory>

int main() {
    int a = 0;
    // Same happens with std::shared_ptr.
    auto a_ptr = std::unique_ptr<int>(&a);
    return 0;
}
```

*** Error in `file': free():
invalid pointer: 0x00007fff30a9a7bc ***

- Create a smart pointer from a pointer to a stack-managed variable
- The variable ends up being owned both by the smart pointer and the stack and gets deleted twice → Error!
// This is a good example of using smart pointers.
#include <iostream>
#include <vector>
#include <memory>
using std::cout; using std::unique_ptr;

struct AbstractShape {  // Structs to save space.
    virtual void Print() const = 0;
};

struct Square : public AbstractShape {
    void Print() const override { cout << "Square\n"; }
};

struct Triangle : public AbstractShape {
    void Print() const override { cout << "Triangle\n"; }
};

int main() {
    std::vector<unique_ptr<AbstractShape>> shapes;
    shapes.emplace_back(new Square);
    auto triangle = unique_ptr<Triangle>(new Triangle);
    shapes.emplace_back(std::move(triangle));
    for (const auto& shape : shapes) { shape->Print(); }
    return 0;
}
Associative containers
std::map

- `#include <map>` to use `std::map`
- Stores **items** under unique keys
- Implemented usually as a **Red-Black tree**
- Key can be any type with operator `<` defined

Create from data:

```cpp
std::map<KeyT, ValueT> m = {
    {key, value}, {key, value}, {key, value}};
```

- Add item to map: `m.emplace(key, value);`
- Modify or add item: `m[key] = value;`
- Get (const) ref to an item: `m.at(key);`
- Check if key present: `m.count(key) > 0;`
- Check size: `m.size();`

http://en.cppreference.com/w/cpp/container/map
std::unordered_map

- `#include <unordered_map>` to use `std::unordered_map`
- Serves same purpose as `std::map`
- Implemented as a **hash table**
- Key type has to be hashable
- Typically used with `int, string` as a key
- Exactly same interface as `std::map`

http://en.cppreference.com/w/cpp/container/unordered_map
Iterating over maps

```cpp
for (const auto& kv : m) {
    const auto& key = kv.first;
    const auto& value = kv.second;
    // Do important work.
}
```

- Every stored element is a pair
- `map` has keys sorted
- `unordered_map` has keys in random order
Type casting
Every variable has a type
Types can be converted from one to another
Type conversion is called type casting
There are 3 ways of type casting:

- static_cast
- reinterpret_cast
- dynamic_cast
static_cast

- Syntax: `static_cast<NewType>(variable)`
- Convert type of a variable at compile time
- **Rarely needed to be used explicitly**
- Can happen implicitly for some types, e.g. `float` can be cast to `int`
- Pointer to an object of a Derived class can be **upcast** to a pointer of a Base class
- Enum value can be caster to `int` or `float`
- Full specification is complex!

reinterpret_cast

- Syntax: `reinterpret_cast<NewType>(variable)`
- Reinterpret the bytes of a variable as another type
- We must know what we are doing!
- Mostly used when writing binary data

dynamic_cast

- Syntax: `dynamic_cast<Base*>(derived_ptr)`
- Used to convert a pointer to a variable of Derived type to a pointer of a Base type
- Conversion happens at runtime
- If `derived_ptr` cannot be converted to `Base*` returns a `nullptr`
- **Google-Style** Avoid using dynamic casting

Enumeration classes
### Enumeration classes

- Store an enumeration of options
- Usually derived from `int` type
- Options are assigned consequent numbers
- Mostly used to pick path in `switch`

```cpp
enum class EnumType { OPTION_1, OPTION_2, OPTION_3 };
```

- Use values as:
  - `EnumType::OPTION_1`, `EnumType::OPTION_2`, ...
- **GOOGLE-STYLE** Name enum type as other types, *CamelCase*
- **GOOGLE-STYLE** Name values as constants `kSomeConstant` or in *ALL_CAPS*
```cpp
#include <iostream>
#include <string>
using namespace std;
enum class Channel { STDOUT, STDERR }
void Print(Channel print_style, const string& msg) {
    switch (print_style) {
    case Channel::STDOUT:
        cout << msg << endl;
        break ;
    case Channel::STDERR:
        cerr << msg << endl;
        break;
    default:
        cerr << "Skipping\n";
    }
}
int main() {
    Print(Channel::STDOUT, "hello");
    Print(Channel::STDERR, "world");
    return 0;
}
```
Explicit values

- By default enum values start from 0
- We can specify custom values if needed
- Usually used with default values

```cpp
enum class EnumType {
    OPTION_1 = 10, // Decimal.
    OPTION_2 = 0x2, // Hexadecimal.
    OPTION_3 = 13
};
```
Read/write binary files
Writing to binary files

- We write a **sequence of bytes**
- We must document the structure well, otherwise no one can read the file
- Writing/reading is **fast**
- No precision loss for floating point types
- Substantially **smaller** than ascii-files

**Syntax**

```cpp
1  file.write(reinterpret_cast<char*>(&a), sizeof(a));
```
Writing to binary files

```cpp
#include <fstream> // for the file streams
#include <vector>
using namespace std;

int main() {
    string file_name = "image.dat";
    ofstream file(file_name,
                   ios_base::out | ios_base::binary);
    if (!file) { return EXIT_FAILURE; }
    int r = 2; int c = 3;
    vector<float> vec(r * c, 42);
    file.write(reinterpret_cast<char*>(&r), sizeof(r));
    file.write(reinterpret_cast<char*>(&c), sizeof(c));
    file.write(reinterpret_cast<char*>(&vec.front()), vec.size() * sizeof(vec.front()));
    return 0;
}
```
Reading from binary files

- We read a **sequence of bytes**
- Binary files are not human-readable
- We must know the structure of the contents

**Syntax**

```cpp
1  file.read(reinterpret_cast<char*>(a), sizeof(a));
```
# Reading from binary files

```cpp
#include <fstream>
#include <iostream>
#include <vector>

using namespace std;

int main() {
  string file_name = "image.dat";
  int r = 0, c = 0;
  ifstream in(file_name,
              ios_base::in | ios_base::binary);
  if (!in) { return EXIT_FAILURE; }
  in.read(reinterpret_cast<char*>(&r), sizeof(r));
  in.read(reinterpret_cast<char*>(&c), sizeof(c));
  cout << "Dim: " << r << " x " << c << endl;
  vector<float> data(r * c, 0);
  in.read(reinterpret_cast<char*>(&data.front()),
           data.size() * sizeof(data.front()));
  for (float d : data) { cout << d << endl; }
  return 0;
}
```